TABLE OF CONTENTS

ALGEBRAIC OPERATIONS WITH VECTORS AND MATRICES

1. Matrix-vector Notation. Matrix-vector Multiplication.

Basic Linear Algebra is about "solving" systems of linear equations.

One general form of this problem is:

given *scalars*

$$\alpha_{ji}, \qquad i = 1, 2, \ldots, m, \quad j = 1,2, \ldots, n,$$

and

$$\beta_j, \qquad j = 1, 2, \ldots, n,$$

find *scalars*

$$\xi_1, \xi_2, \ldots, \xi_m$$

so that

$$\begin{gathered}\alpha_{11}\xi_1 + \alpha_{12}\xi_2 + \ldots + \alpha_{1m}\xi_m = \beta_1\\ \alpha_{21}\xi_1 + \alpha_{22}\xi_2 + \ldots + \alpha_{2m}\xi_m = \beta_2\\ \cdots\cdots\\ \alpha_{n1}\xi_1 + \alpha_{n2}\xi_2 + \ldots + \alpha_{nm}\xi_m = \beta_n.\end{gathered}$$

This is a system of n linear equations in the m unknowns $\xi_1, \xi_2, \ldots, \xi_m$.

We write it, much more compactly, as

$$Ax = b.$$

Here

$$A = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1m} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2m} \\ \vdots & \vdots & & \vdots \\ \alpha_{n1} & \alpha_{n2} & \cdots & \alpha_{nm} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \ldots & a_m \end{bmatrix} = \begin{bmatrix} \alpha_{ji} \end{bmatrix}$$

is the $n \times m$ *coefficient matrix* of the linear system. Its *columns*

$$a_i = \begin{bmatrix} \alpha_{1i} \\ \alpha_{2i} \\ \vdots \\ \alpha_{ni} \end{bmatrix}, \quad i = 1, 2, \ldots, m,$$

and the *right side*

$$b = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

are n-vectors, and

$$x = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_m \end{bmatrix}$$

is the m-vector of unknowns.

The first way of writing a linear system is at the "lowest level," the second is at the "highest level." Note well our *notational conventions.*

Scalars ("numbers") are denoted by lower case Greek letters. This way, if you see a Greek letter you *know* it represents a *scalar.* The scalars can come from a general "field" $\mathbb{F}$. We typically have $\mathbb{F} = \mathbb{R}$, the *real numbers*, but the case $\mathbb{F} = \mathbb{C}$ of *complex numbers* is also important (for "eigenproblems"). Less important, but also arising in applications, are the *rational numbers* $\mathbb{Q}$ and the *integers* $\mathbb{Z}_p$ *modulo* p, a prime number (the case $p = 2$ is used extensively in *algebraic coding theory*).

To keep things simple let us henceforth use $\mathbb{F}$ to denote either $\mathbb{R}$ or $\mathbb{C}$, but be aware that many of our results will extend easily to more general "fields."

An m-vector is an m-*tuple* of scalars, is *always* written as a *column*, and is denoted by a lower case Latin letter. The elements of m-vectors are denoted by a "corresponding" Greek letter. The "correspondence" is imperfect; see the table on the next page. For instance ξ_j is the jth element of x and η_j is the jth element of y. The set of m-vectors with elements from $\mathbb{F}$ is denoted by $\mathbb{F}^m$. Thus the vectors $x, y \in \mathbb{F}^m$ are *equal* if

$$\xi_j = \eta_j, \quad j = 1, 2, \ldots, m .$$

A *very good*, "intermediate level," way to think of an $n \times m$ matrix A is as an m-*tuple of its columns:*

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix}, \quad \text{each } a_j \in \mathbb{F}^n.$$

Of course we may also think of

$$A = \begin{bmatrix} \alpha_{ji} \end{bmatrix}$$

The Greek Alphabet and Latin Notational Correspondents

Alpha	A	α	a	Nu	N	ν	n, v
Beta	B	β	b	Xi	Ξ	ξ	x
Gamma	Γ	γ	c, g	Omicron	O	o	o
Delta	Δ	δ	d	Pi	Π	π	p
Epsilon	E	ϵ	e	Rho	P	ρ	r
Zeta	Z	ζ	z	Sigma	Σ	σ	s
Eta	H	η	h, y	Tau	T	τ	t
Theta	Θ	θ		Upsilon	Y	υ	u
Iota	I	ι	i	Phi	Φ	ϕ	f
Kappa	K	κ	k	Chi	X	χ	
Lambda	Λ	λ	l, ℓ	Psi	Ψ	ψ	
Mu	M	μ	m, u	Omega	Ω	ω	w

as being built from its elements. The scalar α_{ji} lies at the intersection of the ith column and jth *row* of A. With a few basic exceptions matrices are denoted by upper case Latin letters with their columns being denoted by the corresponding lower case Latin letter. Thus a_i is the ith column of A and b_i is the ith column of B. The set of all $n \times m$ matrices with elements from $\mathbb{F}$ is denoted by $\mathbb{F}^{n\times m}$. Thus the matrices of A, B $\in \mathbb{F}^{n\times m}$ are *equal* if

$$a_i = b_i, \qquad i = 1, 2, \ldots, m,$$

that is if

$$\alpha_{ji} = \beta_{ji}, \qquad i = 1, 2, \ldots, m, \quad j = 1, 2, \ldots, n\ .$$

We have *no notation* for the rows of a matrix. If *we* wish to refer to the rows of a matrix we must use transpose notation adroitly–see below.

In *matlab*,

$$\begin{aligned} A(j,i) &= (j,i) \text{ element of } A \\ &= \alpha_{ji}, \\ A(:,i) &= i\text{th column of } A \\ &= a_i, \\ A(j,:) &= j\text{th row of } A \end{aligned}$$

and

$$A(:) = a := \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \in F^{mn},$$

the "supervector" obtained by "stacking" the columns of A to form a single mn-vector. The matrix A is actually stored in the machine as the vector a.

Of course a matrix can consist of one column (or row) and an n-vector can consist of one element. For algebraic purposes n-vectors can be though of as one column matrices.

It turns out that "solving $Ax = b$" involves "factoring" the matrix A in various ways. Before we factor matrices we must learn how to multiply them. But the computation of a matrix product AB is easily reduced to the special case of computing the matrix-vector product $y = Ax$. In fact this is the *most fundamental operation of the whole subject.* Moreover, if we can compute Ax we can check if a tentative solution vector x solves $Ax = b$. How to compute Ax is *evident* from the treatment on page 1.

Matrix-vector multiplication,

$$Ax = \begin{bmatrix} a_1 & a_2 & \dots & a_m \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_m \end{bmatrix}$$

$$= a_1\xi_1 + a_2\xi_2 + \dots + a_m\xi_m$$

$$=: \sum_1^m a_i \, \xi_i$$

$$= \xi_1 a_1 + \xi_2 a_2 + \dots + \xi_m a_m$$

$$=: \sum_1^m \xi_i a_i \, ,$$

is a *linear combination* (ℓc) of the columns of A.

Example.

$$Ax = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 2 \\ 3 & 1 & 0 \\ 4 & 0 & 7 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} 2 + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} (-1) + \begin{bmatrix} -1 \\ 2 \\ 0 \\ 7 \end{bmatrix} 3$$

$$= \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ -1 \\ 0 \end{bmatrix} + \begin{bmatrix} -3 \\ 6 \\ 0 \\ 21 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \\ 3 \\ 5 \\ 8 \end{bmatrix} + \begin{bmatrix} -3 \\ 6 \\ 0 \\ 21 \end{bmatrix} = \begin{bmatrix} -1 \\ 9 \\ 5 \\ 29 \end{bmatrix}$$

Thus

$$x = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$$

solves

$$Ax = b := \begin{bmatrix} -1 \\ 9 \\ 5 \\ 29 \end{bmatrix}.$$

Of course this is the *same* as the "usual" way of computing $y = Ax$ which is

$$\eta_j = \sum_{i=1}^{m} \alpha_{ji}\xi_i , \qquad j = 1, 2, \ldots, n.$$

If μ_{F} (α_{F}) is a floating point multiplication (addition) of elements of F then the flop count for computing $y = Ax$, with $A \in \mathsf{F}^{n\times m}$, is $mn\,\mu_{\mathsf{F}} + (m-1)\,n\,\alpha_{\mathsf{F}}$. For *practical* problems m and n can be *very large*; in this case this is about mn $(\mu_{\mathsf{F}} + \alpha_{\mathsf{F}})$. But our "vector formulation" can be much faster on machines with "vector architectures."

But what is more important is the *link* between Ax, the key operation of matrix computation, and *linear combination*, the key concept of (the more abstractly oriented subject of) Linear Algebra.

Linear combinations of elements of F^n are built up from the two more basic operations of *addition* and *scalar multiplication*:

a. if $x, y \in \mathsf{F}^n$ then

$$x + y = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix} + \begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{bmatrix}$$

$$:= \begin{bmatrix} \xi_1 + \eta_1 \\ \xi_2 + \eta_2 \\ \vdots \\ \xi_n + \eta_n \end{bmatrix} \equiv \begin{bmatrix} \eta_1 + \xi_1 \\ \eta_2 + \xi_2 \\ \vdots \\ \eta_n + \xi_n \end{bmatrix}$$

$$\equiv y + x;$$

b. if $x \in \mathsf{F}^n$, $\alpha \in \mathsf{F}$, then

$$x\alpha = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix} \alpha := \begin{bmatrix} \xi_1\alpha \\ \xi_2\alpha \\ \vdots \\ \xi_n\alpha \end{bmatrix}$$

$$\equiv \begin{bmatrix} \alpha\xi_1 \\ \alpha\xi_2 \\ \vdots \\ \alpha\xi_n \end{bmatrix} =: \alpha x .$$

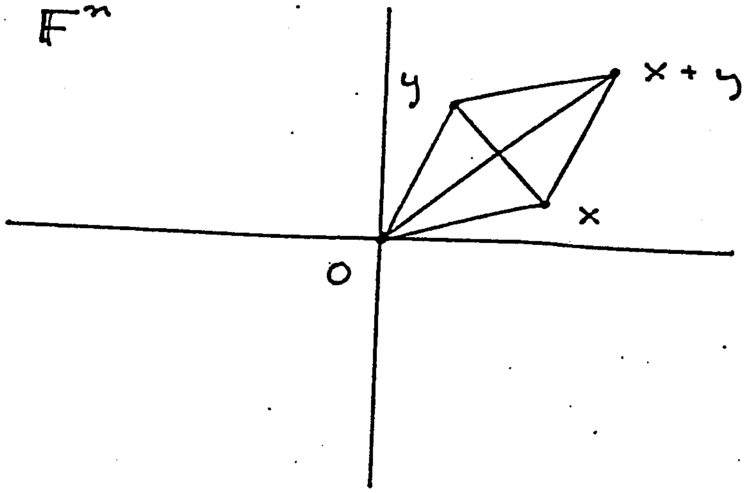
The fact that $x+y \equiv y+x$ follows from the commutativity of addition in F. Since multiplication in F is also commutative we also write *either* αx or $x\alpha$ for scalar multiplication. The first of these is more usual in Linear Algebra but the latter is most often more suggestive. For instance it is certainly more natural in our definition of Ax.

The *zero element* in F^n is

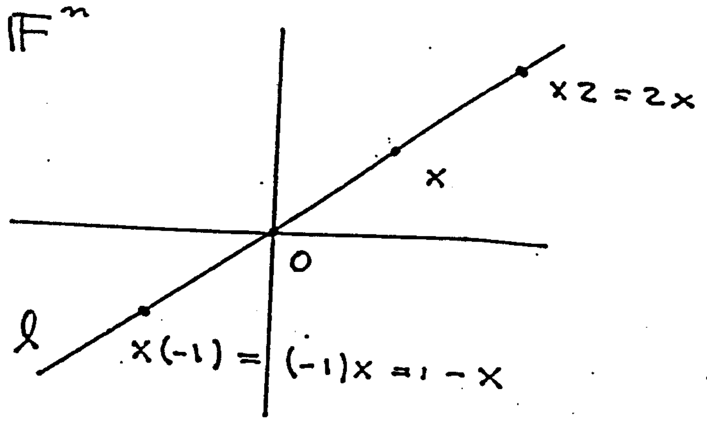
$$0 := 0_n := \left.\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}\right\} \text{n elements} .$$

Geometric interpretations:

a.



b.



a is the "parallelogram rule" for adding elements of F^n. These pictures are merely *suggestive*. You can *see* them in R^2, R^3 and even in R^n for general n. In R^n three vectors usually determine a *plane* and two vectors usually determine a *line*. In $\mathbf{R}^n$, if $x \neq 0$, as α runs over all of $\mathbf{R}$ the point $x\alpha = \alpha x$ runs over the whole *line* ℓ passing through 0 and x. For $\mathbf{R} = \mathbf{C}$ use your "imagination."

Note that the formation of ℓcs of elements of $\mathbf{F}^n$ is *trivial*: the algebraic operations are merely done *elementwise* ($y = Ax = \sum_1^m a_i\xi_i$).

The set $\mathbf{F}^n$, together with the operations of addition and scalar multiplication, is the *primary* and *most important* example of an algebraic system known as a *linear space* (or a vector space). More precisely, $\mathbf{F}^n$ is a *linear space with respect to the "field"* $\mathbf{F}$. (Not so) roughly speaking, a general *linear space* is an algebraic system in which it "makes sense" to form ℓcs. There is a set of elements $\mathcal{V}$ and a scalar "field" $\mathbf{F}$, usually $\mathbf{F} = \mathbf{R}$ or $\mathbf{F} = \mathbf{C}$. One must be able to form ℓcs of *finitely many* elements of $\mathcal{V}$, with coefficients from $\mathbf{F}$. The *key requirement* is that these ℓcs must again be in $\mathcal{V}$. This is *closure* under the formation of ℓcs.

The (many) *axioms* for a general linear space are *simple* abstractions of the basic properties of the linear space $\mathbf{F}^n$. They are tedious and not very interesting. We *may* put them into an appendix. But there are plenty of interesting and useful examples of linear spaces which are related with matrices.

In fact $\mathcal{V} = \mathbf{F}^{n\times m}$ is a linear space with respect to $\mathbf{F}$. If

$$A = \left[\alpha_{ji}\right] \quad \text{and} \quad B = \left[\beta_{ji}\right]$$

are in $\mathbf{F}^{n\times m}$ then the *sum* of A and B is

$$\begin{aligned} A + B :&= \left[\alpha_{ji} + \beta_{ji}\right] \\ &= \left[\beta_{ji} + \alpha_{ji}\right] = B + A \end{aligned}$$

and the *multiple* of A by the *scalar* $\alpha \in \mathbf{F}$ is

$$\begin{aligned} \alpha A :&= \left[\alpha\alpha_{ji}\right] \\ &= \left[\alpha_{ji}\alpha\right] =: A\alpha. \end{aligned}$$

Again, the algebraic operations are merely done elementwise. The *key* assertion of *closure*, that $A + B$ and αA are again $n \times m$ matrices with elements in $\mathbf{F}$, is *trivial.*

Note that we form ℓcs of $n \times m$ matrices in *algebraically the same way* as we form ℓcs of their associated "supervectors." In fact matlab forms ℓcs of matrices this way.

There is an operation of "scalar addition" which is peculiar to matlab, but does not occur in usual matrix theory. If $A = \left[\ \alpha_{ji}\ \right] \in \mathbb{F}^{n\times m}$ and $\alpha \in \mathbb{F}$ then

$$\alpha + A := \left[\alpha + \alpha_{ji}\right].$$

Thus α is added to each element of A.

2. Linear transformations.

Matrix-vector multiplication,

$$\begin{aligned} A : \mathbb{F}^m &\to \mathbb{F}^n \\ x &\to Ax \end{aligned}$$

is a linear transformation, that is

$$A(\alpha x + \beta y) \equiv \alpha(Ax) + \beta(Ay)$$

(identically, for *all* $x, y \in \mathbb{F}^m$ and *all* $\alpha, \beta \in \mathbb{F}$).

$\square$. By the definitions,

$$\begin{aligned} A(\alpha x + \beta y) &:= \sum_1^m a_j(\alpha\xi_j + \beta\eta_j) \\ &\equiv \alpha \sum_1^m a_j\xi_j + \beta \sum_1^m a_j\eta_j \\ &=: \alpha(Ax) + \beta(\overset{A}{\cancel{B}}y). \end{aligned}$$

✓

$\blacksquare$

Problem 1. Which way is cheaper, in terms of flops, for *large* m and n?

Definitions.

a. An (abstractly given) function

$$\begin{aligned} \mathcal{A} : \mathbb{F}^m &\to \mathbb{F}^n \\ x &\to \mathcal{A}(x) \end{aligned}$$

is a *linear transformation* if

$$\mathcal{A}(\alpha x + \beta y) \equiv \alpha\mathcal{A}(x) + \beta\mathcal{A}(y).$$

b. The columns of the $n \times n$ *identity matrix*

$$I = I_n = \left[\ e_1\ \ e_2\ \ \dots\ \ e_n\ \right]$$

$$:= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (n = 4)$$

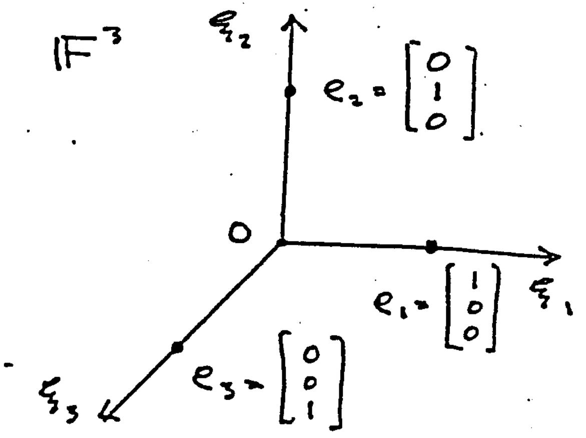
are the *axis vectors* in F^n. Their "lengths" (here $n = 4$), not exhibited in the notation, are determined by the context. The set $\{e_1, e_2, \ldots, e_n\}$ forms the *standardbasis* for F^n: *every* $x \in F^n$ can be expressed *uniquely* as the ℓc

$$x = \sum_1^n e_i \xi_i .$$

In matrix terms this is just $x = Ix$!

The word "basis" is also a key word of Linear Algebra but we are not using it here; "standardbasis" is *one word* and we have just defined the standardbasis for F^n to be $\{e_1, e_2, \ldots, e_n\}$.

The axis vectors lie on the coordinate axes. For $n = 3$:


$\mathbb{F}^3$
ξ_2
$e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$
0
$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$
ξ_1
$e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$
ξ_3

We have

$$\boxed{Ae_i \equiv a_i}$$

the ith column of A.

Every linear transformation

$$\mathcal{A}:\ F^m \to F^n$$

can be represented by a matrix. In fact we have

$$\mathcal{A}(x) = Ax \quad \text{with} \quad a_i \equiv \mathcal{A}(e_i).$$

□ Write x as an ℓc of the axis vectors and use the linearity of $\mathcal{A}$ (this works for ℓcs with arbitrary finite numbers of terms):

$$\mathcal{A}(x) \equiv \mathcal{A}\left(\sum_1^m e_i \xi_i\right)$$
$$\equiv \sum_1^m \underbrace{\mathcal{A}(e_i)}_{=:\, a_i} \xi_i$$
$$= Ax.$$ ■

Pure mathematicians use this in the reverse direction, to forget about matrices. But applied scientists *must* compute with matrices!

Example. Let the "abstract" transformation $\mathcal{R}_\theta$ *rotate* each vector in $\mathbf{R}^2$ counterclockwise through the angle θ. To see that $\mathcal{R}_\theta$ is linear note that $\mathcal{R}_\theta$ rotates each of the figures on page 7 ($\mathbf{F}^n := \mathbf{R}^2$) similarly. Thus

$$\mathcal{R}_\theta(x+y) \equiv \mathcal{R}_\theta(x) + \mathcal{R}_\theta(y),$$
$$\mathcal{R}_\theta(\alpha x) \equiv \alpha\mathcal{R}_\theta(x),$$

and so $\mathcal{R}_\theta$ is linear. But what is its matrix representation? By Trigonometry,

$$\mathcal{R}_\theta(e_1) = \begin{bmatrix} \gamma \\ \sigma \end{bmatrix}, \quad \mathcal{R}_\theta(e_2) = \begin{bmatrix} -\sigma \\ \gamma \end{bmatrix},$$

with

$$\gamma := \cos\theta, \quad \sigma := \sin\theta.$$

Thus

$$R_\theta = \begin{bmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{bmatrix}$$

represents $\mathcal{R}_\theta$. Note that e_1 and e_2 are perpendicular, and so are their "$\mathcal{R}_\theta$-transforms," the columns of R_θ.

3. Matrix multiplication = composition of linear transformations.

In general, in Mathematics, if we have two functions f and g, with the range of g contained in the domain of f, then we may *compose* f and g to get

$$\begin{tikzcd}\mathcal{D} \arrow[r, bend left, "g"] \arrow[rr, bend right] & \mathcal{E} \arrow[r, bend left, "f"] & \mathcal{F}\end{tikzcd}$$

$$h = f \circ g$$

with

$$h(x) := f(g(x)), \quad x \in \mathcal{D}.$$

Now consider the *matrix-vector multiplications*

$$\begin{tikzcd}\mathbb{F}^m \arrow[r, bend left, "B"] \arrow[rr, bend right] & \mathbb{F}^p \arrow[r, bend left, "A"] & \mathbb{F}^n\end{tikzcd}$$

$$\mathcal{C}(x) := A(Bx) .$$

The composed transformation $\mathcal{C}$ is linear, for we have

$$\begin{aligned}\mathcal{C}(\alpha x + \beta y) &:= A[B(\alpha x + \beta y)] \\ &\equiv A[\alpha Bx + \beta By] \\ &\equiv \alpha A(Bx) + \beta A(By) \\ &=: \alpha \mathcal{C}(x) + \beta \mathcal{C}(y),\end{aligned}$$

using the linearity of each matrix-vector multiplication. Thus $\mathcal{C}$: $\mathbb{F}^m \to \mathbb{F}^n$ is represented by the matrix

$$C = \begin{bmatrix} c_1 & c_2 & \dots & c_m \end{bmatrix}$$

in which

$$c_j \equiv C(e_j) : \equiv A(Be_j) \equiv Ab_j.$$

The matrix $C \in F^{n \times m}$ is called the *matrix product* of

$$A \in F^{n \times p} \text{ and } B \in F^{p \times m},$$

in this order, and we write

$$AB := C.$$

We have thus shown the very important relation that

$$\boxed{\begin{aligned} AB &= A\begin{bmatrix} b_1 & b_2 & \dots & b_m \end{bmatrix} \\ &= \begin{bmatrix} Ab_1 & Ab_2 & \dots & Ab_m \end{bmatrix} \end{aligned}}$$

That is the ith column of the matrix product AB is A times the ith column of B. This *reduces* the problem of computing AB to that of computing Ax, with x running through all the columns of B.

Problem 2. What is the flop count for matrix multiplication, with A and B as above?

Of course we still have the "lowest level" formulation of matrix multiplication in terms of the elements of the matrices. If $A = \begin{bmatrix} \alpha_{ji} \end{bmatrix}$, $B = \begin{bmatrix} \beta_{ji} \end{bmatrix}$ and $C = \begin{bmatrix} \gamma_{ji} \end{bmatrix}$ (our standard notation) with $C = AB$ then

$$\boxed{\gamma_{ji} = \sum_{k=1}^{p} \alpha_{jk}\beta_{ki}, \qquad i = 1, 2, \dots, m, \quad j = 1, 2, \dots, n.}$$

In fact these are the *same formulas as above*, but written in a clumsier way. Both techniques give the *same results numerically*, provided all sums are computed in the natural order!

Example.

$$AB = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 2 \\ 3 & 1 & 0 \\ 4 & 0 & 7 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ -1 & 1 \\ 3 & 1 \end{bmatrix}.$$

A is 4×3, B is 3×2, so AB is 4×2.

We already computed

$$Ab_1 = \begin{bmatrix} -1 \\ 9 \\ 5 \\ 29 \end{bmatrix}$$

on page 5. And since the elements of b_2 are all ones, Ab_2 is the *sum of the columns of* A. Thus

$$AB = \begin{bmatrix} -1 & 0 \\ 9 & 5 \\ 5 & 4 \\ 29 & 11 \end{bmatrix}.$$

The reader should check this with the usual "two finger" method of matrix multiplication.

Matrix multiplication is not commutative, that is

$$AB \neq BA$$

in general. In fact if

$$A \in F^{n\times q} \text{ and } B \in F^{p\times m}$$

then AB is defined only if $p = q$, and is $n \times m$, BA is defined only if $m = n$, and is $p \times q$. Thus AB and BA are *both* defined and are *comparable* if and only if $m = n = p = q$. That is both matrices must be square and of the same *order* n. But almost any pair of 2×2 matrices one writes down will not commute. For example

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \; B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

give

$$AB = A\begin{bmatrix} e_2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},$$

$$BA = B\begin{bmatrix} 0 & e_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \neq AB.$$

Matrix multiplication is *associative*, that is

$$A(BC) \equiv (AB)C,$$

provided these matrix multiplications make sense. This would be very tedious to verify directly, but it follows from the general fact that composition of (any kinds of) functions is always associative. The distributive law

$$A(B+C) \equiv AB + AC$$

follows from the linearity of Ax in the vector x:

$$A(b_i + c_i) \equiv Ab_i + Ac_i.$$

The distributive law

$$(A+B)C \equiv AC + BC$$

follows from the linearity of Ax in the matrix A:

$$(A+B)\,c_i \equiv Ac_i + Bc_i\,.$$

Extended operations with matrices are easy. The matrix additions and multiplications must be well-defined and one must be careful not to commute them under multiplication. But such computations are rarely done. As we have said the main problem of basic Linear Algebra is to solve systems of linear equations, and this is done by factoring a single matrix.

We give *matlab codes* for the two methods of matrix multiplication which we discussed above. We first give the "usual" "two finger" method. We compute $C = AB$.

```
function C = mmul0(A, B)
[n q] = size (A);
[p m] = size (B);
if p ~= q
    error ('AB not defined.')
end
C = zeros(n, m);
for i = 1 : m
    for j = 1 : n
        s = 0;
        for k  = 1 : p
            s = s + A(j, k) * B(k, i);
        end
        C(j, i) = s;
        end
    end
```

This is the usual way to code matrix multiplication, in fortran say. It involves a *triple loop* and loops are *incredibly slow* in matlab. The statement C = zeros(n, m) sets aside a storage area for C. It is not strictly necessary but it makes the triple loop as fast as possible.

By contrast the column oriented code involves *only one loop*:

```
function C = mmul1(A, B)
[n q] = size (A);
[p m] = size (B);
if p ~= q
    error ('AB not defined.')
end
C = zeros (n,m);
for i = 1 : m
    C(:, i) = A * B(:, i);
end
```

Of course the highest level code is the single command C = A*B!

All three codes *should* give the *same results, numerically*, on Suns and HPs.

Problem 3. Do they, in fact, give the same numerical results on these machines?

4. Block multiplication

For any matrix A let

$$\text{cols } A: = \#\text{ columns of } A,$$
$$\text{rows } A: = \#\text{ rows of } A.$$

Following is a general result on *block multiplication* which generalizes substantially the "two finger" method of matrix multiplication.

If A and B are *partitioned into blocks,*

$$A = \left[A_{ji} \right], \qquad i = 1, 2, \dots, p, \quad j = 1, 2, \dots, n,$$
$$B = \left[B_{ji} \right], \qquad i = 1, 2, \dots, m, \quad j = 1, 2, \dots, p,$$

so that

$$\text{cols } A_{jk} \equiv \text{rows } B_{ki}$$

then

$$AB = \left[C_{ji} \right], \quad i = 1, 2, \ldots, m, \quad j = 1, 2, \ldots, n,$$

with

$$C_{ji} = \sum_{k=1}^{p} A_{jk} B_{ki} .$$

□ The proof of this basic fact is "too boring for words." We will use only very special cases of it. In fact we have already worked with the partitioning of B by columns and, for the one column matrix B = x, the partitioning of B by rows. ■

Example. For partitioning A and B into 2×2 block matrices we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

provided the eight matrix multiplications are defined. Thus

$$AB = \left[\begin{array}{c|cc} 1 & 1 & 0 \\ \hline 2 & 0 & 1 \\ 3 & 1 & 0 \end{array} \right] \left[\begin{array}{c|cc} 1 & 1 & 0 \\ \hline 1 & 2 & 3 \\ 1 & 3 & 2 \end{array} \right] = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

with

$$C_{11} = 1 \cdot 1 + \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1 + 1 = 2,$$

$$C_{21} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} 1 + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix},$$

$$C_{12} = 1 \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 3 \end{bmatrix},$$

$$C_{22} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 0 \\ 3 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 5 & 3 \end{bmatrix}.$$

So

$$AB = \left[\begin{array}{c|cc} 2 & 3 & 3 \\ \hline 3 & 5 & 2 \\ 4 & 5 & 3 \end{array}\right]$$

as can be checked with the "two finger" method.

Application. The basic step of Gauss factorization. If

$$A = \begin{bmatrix} \alpha & b' \\ a & B \end{bmatrix}, \quad \alpha \neq 0,$$

then

$$A = \begin{bmatrix} 1 & \\ \ell & I \end{bmatrix} \begin{bmatrix} \alpha & b' \\ & G \end{bmatrix}$$

with

$$\ell = \frac{a}{\alpha}, \quad G = B - \ell b'.$$

Here b' is the (conjugate) transpose of b. See §5. The result is *easily* verified by *block multiplication.*

Problem 4. If the two 4×4 matrices A and B are partitioned as

$$A = \left[\begin{array}{c|cc|c} x & x & x & x \\ x & x & x & x \\ \hline x & x & x & x \\ x & x & x & x \end{array}\right] \qquad B = \left[\begin{array}{c|ccc} x & x & x & x \\ \hline x & x & x & x \\ x & x & x & x \\ \hline x & x & x & x \end{array}\right]$$

then what is the *block structure* of $C = AB$? I.e., what are the *sizes* of the blocks?

If we partition A by columns and B by rows then we can compute AB as a sum of "outer products." See §5.

In matlab:

```
function C = mmul2(A, B)
[n q] = size (A);
[p m] = size (B);
if p  ~= q
    error('AB not defined.')
end
C = zeros (n, m);
for k = 1 : p
    C = C + A(:, k) * B(k, :);
end
```

This form of matrix multiplication is used in our codes mulm··· for computing *doubled precision* matrix products.

Cost of computing AB, A, B $\in F^{n \times n}$.

$$\begin{aligned} Ax \text{ uses } \mu(n) &= n^2 \text{ mults} \\ \text{and } \alpha(n) &= n(n-1) \text{ adds.} \end{aligned}$$

Thus $AB = \begin{bmatrix} Ab_1 & Ab_2 & \dots & Ab_n \end{bmatrix}$, computed in the "naive" way, uses

$$\begin{aligned} \mu(n) &= n^3 \text{ mults, and} \\ \alpha(n) &= n^2(n-1) \text{ adds.} \end{aligned}$$

For n = 2, multiplication of two 2×2 matrices, we have

$$\mu(2) = 8, \;\; \alpha(2) = 4.$$

Application of block multiplication. Fast matrix multiplication (*Strassen* 1969).

a. If

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \doteq AB$$

then C can be computed from

$$\begin{aligned} C_{11} &= D_1 + D_4 - D_5 + D_7, \\ C_{21} &= D_2 + D_4, \\ C_{12} &= D_3 + D_5 \\ C_{22} &= D_1 + D_3 - D_2 + D_6, \end{aligned}$$

with

$$D_1 := (A_{11} + A_{22})(B_{11} + B_{22}),$$
$$D_2 := (A_{21} + A_{22})\, B_{11},$$
$$D_3 := A_{11}\, (B_{12} - B_{22}),$$
$$D_4 := A_{22}\, (B_{21} - B_{11}),$$
$$D_5 := (A_{11} + A_{12})\, B_{22},$$
$$D_6 := (A_{21} - A_{11})\, (B_{11} + B_{12}),$$
$$D_7 := (A_{12} - A_{22})\, (B_{21} + B_{22}).$$

b. Corollary. Two 2×2 matrices can be multiplied with 7 multiplications and 18 additions of scalars. Thus one multiplication can be replaced by 14 additions. This is not a good trade on today's machines, or even on older ones.

c. Better corollary. The product of two $2n \times 2n$ matrices can be computed with 7 multiplications and 18 additions of $n \times n$ matrices.

d. Now *divide* and *conquer*. For $n = 2^k$, $k = 0, 1, 2, \ldots$, let $\mu(n)$ and $\alpha(n)$ denote, respectively, the numbers of multiplications and additions *of scalars* used to multiply two $n \times n$ matrices. Thus, by c, we have the *recurrence relations*

$$\mu(1) = 1, \quad \alpha(1) = 0,$$
$$\mu(2n) = 7\,\mu(n),$$
$$\alpha(2n) = 7\,\alpha(n) + 18n^2 .$$

By induction, or direct verification,

$$\mu(n) = 7^k, \quad k = \log_2 n,$$
$$\alpha(n) = 6\left(7^k - 4^k\right) < 6 \cdot 7^k.$$

Now

$$7^k = \left(2^{\log_2 7}\right)^k = \left(2^k\right)^{\log_2 7} = n^\beta$$

with

$$\boxed{\beta := \log_2 7 \doteq 2.8074.}$$

Thus we have

Strassen's theorem. Multiplication of two $n \times n$ matrices, with $n = 2^k$, requires at most n^β multiplications and $6n^\beta$ additions of scalars.

Kolmogorov conjectured that the *exponent* for matrix multiplication, here $\beta \doteq 2.8$, could be reduced to 2! Pan, Schönhage reduced it to 2.51, roughly. It may now be < 2.5, but the associated algorithms must not be very elegant.

Problem 5.

a. Verify Strassen's identities of part (a) above.

b. Show that $\mu(n) = 7^k$, $\alpha(n) = 6\left(7^k - 4^k\right)$, with $k = \log_2 n$, and equivalently $n = 2^k$, solve the recurrence relations of part d above.

Our matlab code *strassen* follows. It turns out to be *incredibly* slow in matlab!!

```
function C = strassen(A, B)

C is the product of the n by n matrices A and B computed by STRASSEN's fast matrix
multiplication algorithm. This program uses recursive calls. To keep the code simple it is
assumed that n is a power of two (n = 1, 2, 4, 8, ... ).

Strassen's algorithm multiplies two matrices of order n = 2^k with 7^k multiplications and
6(7^k - 4^k) additions. The total arithmetic work is less than 7(7^k) = 7n^lg 7 flops. Here lg
is the base two logarithm. Since lg 7 .=. 2.8074 this compares favorably with 2n^3 flops for the
usual method.

Copyright (c) 8 September 1990 by Bill Gragg. All rights reserved. Revised 25 March 1994.
strassen calls strassen.

begin strassen

   Perform some error checks.

   [n m] = size(A);

   if m ~= n
       error ('First matrix is not square.')
   end

   [n m] = size(B);

   if m ~= n
       error('Second matrix is not square.')
   end

   m = n;

   while m > 1
       m = m/2;
   end

   if m ~= 1
       error ('Order of matrices is not a power of two.')
   end
```

```
    "Do Strassen".
    if $n < 2$
        The trivial case $n = 1$.
        $C = A*B$;
    else
        Partition the matrices. (This creates additional storage!)
        $m = n/2$;        $p = 1:m$;        $q = m+1:n$;
        $A11 = A(p,p)$;   $A12 = A(p,q)$;   $B11 = B(p,p)$;   $B12 = B(p,q)$;
        $A21 = A(q,p)$;   $A22 = A(q,q)$;   $B21 = B(q,p)$;   $B22 = B(q,q)$;
        Use recursive calls to compute seven matrix products, also using ten matrix additions.
        $D1 = strassen(A11+A22,B11+B22)$;   $D2 = strassen(A21+A22,B11)$;
        $D3 = strassen(A11,B12-B22)$;       $D4 = strassen(A22,B21-B11)$;
        $D5 = strassen(A11+A12,B22)$;       $D6 = strassen(A21-A11,B11+B12)$;
        $D7 = strassen(A12-A22,B21+B22)$;
        Now compute the blocks of C, using eight more matrix additions.
        $C11 = D1 + D4 - D5 + D7$;     $C12 = D3 + D5$;
        $C21 = D2 + D4$;               $C22 = D1 - D2 + D3 + D6$;
        Arrange the blocks to form C itself.
        $C = [C11\ \ C12;\ \ C21\ \ C22]$;
    end
end strassen
```

Reference:

[1] Volker Strassen, Gaussian elimination is not optimal. Numer. Math. 13(1969) 354-356.

5. Transposition and conjugate transposition. Euclidean inner products and outer products. Euclidean norm. Orthogonality.

Transposition.

The *transpose* of the column n-vector

$$x = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix}$$

is the *row* n-vector

$$x^T := \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_n \end{bmatrix}.$$

The *transpose* of the $n \times m$ matrix

$$A = \begin{bmatrix} a_1 & a_2 & \dots & a_m \end{bmatrix} = \begin{bmatrix} \alpha_{ji} \end{bmatrix}$$

is the $m \times n$ matrix

$$A^T := \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix} = \begin{bmatrix} \alpha_{ij} \end{bmatrix}.$$

Example.

$$\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}.$$

Thus A^T is the *reflection* of A in its *main diagonal* $\alpha_{11}, \alpha_{22}, \alpha_{33}, \dots$. Matlab uses "'" for "T." Since the matrix A represents a linear transformation from $\mathbb{F}^m$ to $\mathbb{F}^n$ then A^T represents a linear transformation from $\mathbb{F}^n$ to $\mathbb{F}^m$. (This is not the "inverse" of A. Matrix transposition is *much simpler* than "matrix inversion.")

If we transpose A^T we get A back:

$$A \equiv (A^T)^T.$$

In particular for the one column matrix x,

$$x \equiv (x^T)^T = \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_n \end{bmatrix}^T.$$

This is a typographically more convenient way to write (column) vectors.

We have defined A^T through the partitioning of A by its columns. For more general partitionings we have results like: if

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

then

$$A^T = \begin{bmatrix} A^T_{11} & A^T_{21} & A^T_{31} \\ A^T_{12} & A^T_{22} & A^T_{32} \end{bmatrix}$$

Check this out on an example, *not* a 3×2 matrix!

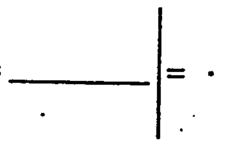
How does transposition relate with the formation of *ℓcs* of matrices? This result is clear:

$$\boxed{(\alpha A + \beta B)^T \equiv \alpha A^T + \beta B^T}$$

A *scalar product* (*dot product* in *old* terminology) is a row times a column:

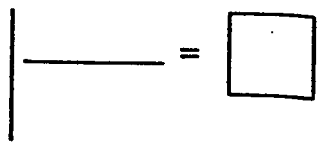
$$\begin{aligned} v^T u &= \begin{bmatrix} \nu_1 & \nu_2 & \dots & \nu_n \end{bmatrix} \begin{bmatrix} \upsilon_1 \\ \upsilon_2 \\ \vdots \\ \upsilon_n \end{bmatrix} \\ &= \nu_1 \upsilon_1 + \nu_2 \upsilon_2 + \dots + \nu_n \upsilon_n \\ &= \sum_1^n \nu_k \upsilon_k \\ &\equiv \sum_1^n \upsilon_k \nu_k \\ &\equiv u^T v. \end{aligned}$$

We can remember this with a picture:

$v^T u =$ 

The *dot* "." represents a *scalar!* Note that u and v must have the same "length" n (matlab's "length"). If $\mathbb{F} = \mathbb{R}$ then $v^T u$ is also the *Euclidean inner product* of u and v, but it is *not* the Euclidean inner product of u and v if either has any nonreal elements.

An *outer product* is a column times a row:

$vu^T =$ 

$$= v\begin{bmatrix} v_1 & v_2 & \dots & v_m \end{bmatrix}$$

$$= \begin{bmatrix} vv_1 & vv_2 & \dots & vv_m \end{bmatrix}$$

Each column of vu^T is a scalar multiple of the *same* vector v. Outer products are basic for a "correct" treatment of Gauss factorization, but they should be computed explicitly, in *elementwise* form, only rarely. After all, if $A = vu^T$ is $n \times m$ then it takes mn storage locations to store the elements of A, but only $m+n$ locations to store the vectors u and v which determine A.

Matrices like

$$A = I + vu^T$$

occur in important applications and they are used to compute vectors

$$y = Ax = (I + vu^T)x$$

$$= x + v\alpha, \quad \alpha = u^T x.$$

If A is $n \times n$, that is if $u, v \in \mathbb{F}^n$, then the indicated algorithm computes y with about $2n\ (\mu_{\mathbb{F}} + \alpha_{\mathbb{F}})$ for *large* n. By contrast, the flop count for computing Ax when A is an *arbitrary* $n \times n$ matrix is $n^2\ (\mu_{\mathbb{F}} + \alpha_{\mathbb{F}})$.

Problem 6. Show that a product of two outer products is a scalar product times an outer product.

To describe the "usual" "two finger" method of matrix multiplication, since we have no notation for the rows of a matrix, we write the matrix product as

$$A^T B = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_n^T \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \dots & b_m \end{bmatrix}$$

$$= \begin{bmatrix} a_j^T b_i \end{bmatrix} \qquad i = 1, 2, \dots, m,\ j = 1, 2, \dots, n\ .$$

This is the *scalar product form* of matrix multiplication. Of course it is another special instance of block multiplication. But we also have the more interesting *outer product form* of matrix multiplication

$$AB^T = \begin{bmatrix} a_1 & a_2 & \dots & a_p \end{bmatrix} \begin{bmatrix} b_1^T \\ b_2^T \\ \vdots \\ b_p^T \end{bmatrix}$$

$$= a_1 b_1^T + a_2 b_2^T + \ldots + a_p b_p^T$$

$$=: \sum_1^p a_k\, b_k^T .$$

The most important fact about transposition is the *reverse order rule for transposition:*

$$\boxed{(AB)^T \equiv B^T A^T}\,.$$

This can be remembered with the help of a picture:

$$\begin{tikzcd}
\mathbb{F}^m \arrow[r, bend left, "B"] \arrow[rr, bend left=60, "AB"] & \mathbb{F}^p \arrow[l, bend left, "B^T"] \arrow[r, bend left, "A"] & \mathbb{F}^n \arrow[l, bend left, "A^T"] \arrow[ll, bend left=60, "(AB)^T"]
\end{tikzcd}$$

(But *remember*: transposes are *not* "inverses"!)

□ We prove the reverse order rule using the outer product form of matrix multiplication. First, if

$$vu^T = \left[\, \nu_j \upsilon_i \,\right], \qquad i = 1, 2, \ldots, m, \quad j = 1, 2, \ldots, n,$$

then

$$(vu^T)^T = \left[\, \nu_i \upsilon_j \,\right], \qquad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, m,$$

$$= \left[\, \upsilon_j \nu_i \,\right]$$

$$= uv^T .$$

Now consider

$$AB^T = \sum_1^p a_k b_k^T ,$$

a sum of outer products. Since the transpose of a linear combination of matrices is the *same* linear combination of their transposes (the boxed formula on page 24) then

$$\begin{aligned}(AB^T)^T &= \left(\sum_1^p a_k b_k^T\right)^T \\ &= \sum_1^p \left(a_k b_k^T\right)^T \\ &= \sum_1^p b_k a_k^T \\ &= BA^T.\end{aligned}$$

Now replace B^T by B, and B by B^T (i.e., *rename* B^T as B).

■.

Problem 7. Prove the reverse order rule for transposition using the scalar product form of matrix multiplication.

Conjugation.

The conjugate of

$$A = \left[\, \alpha_{ji} \,\right]$$

is

$$\overline{A} := \left[\, \overline{\alpha}_{ji} \,\right] = \left[\, \alpha'_{ji} \,\right]$$

$$= \text{conj(A)} \quad \text{in matlab.}$$

Example. If

$$A = \begin{bmatrix} 1+3i & 4+i \\ 2-4i & -i \\ 3-i & 1+i \end{bmatrix}$$

then

$$\overline{A} = \begin{bmatrix} 1-3i & 4-i \\ 2+4i & i \\ 3+i & 1-i \end{bmatrix}$$

Thus the operation of conjugation is just applied *elementwise* to A to get $\overline{A}$. Of course if $A \in R^{n\times m}$ then $\overline{A} = A$.

Conjugate transposition.

The *conjugate transpose* of

$$A = \left[\alpha_{ji} \right]$$

is

$$A' := \left[\alpha'_{ij} \right] \equiv \overline{A}^T \equiv \overline{A^T} .$$

Thus we may conjugate A and transpose the result or, equivalently, we may transpose A and conjugate the result.

Example. With the above A,

$$\overline{A}^T = \begin{bmatrix} 1-3i & 2+4i & 3+i \\ 4-i & i & 1-i \end{bmatrix},$$

$$A^T = \begin{bmatrix} 1+3i & 2-4i & 3-i \\ 4+i & -i & 1+i \end{bmatrix},$$

$$\overline{A^T} = \begin{bmatrix} 1-3i & 2+4i & 3+i \\ 4-i & i & 1-i \end{bmatrix}$$

$$\equiv \overline{A}^T \equiv: A',$$

in general.

We use A', *as in matlab*, to denote the conjugate transpose of A. The more frequent notation is A^H, and A^* is also used.

The facts that we established above for transposition all extend in straightforward ways to corresponding facts for conjugate transposition. Clearly,

$$A \equiv (A')' .$$

and if

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{33} \end{bmatrix}$$

then

$$A' = \begin{bmatrix} A'_{11} & A'_{21} \\ A'_{12} & A'_{22} \\ A'_{13} & A'_{33} \end{bmatrix}.$$

The previous rule

$$(\alpha A + \beta B)^T \equiv \alpha A^T + \beta B^T,$$

at least for square matrices A, B, states that *transposition is a linear transformation on the linear space* $F^{n \times m}$. For conjugate transposition we have instead

$$(\alpha A + \beta B)' \equiv \alpha' A' + \beta' B'$$
$$= \overline{\alpha} A' + \overline{\beta} B'.$$

For square matrices A, B this says that *conjugate transposition* is a *conjugate linear transformation on the linear space* $C^{n \times n}$.

The *reverse order rule for conjugate transposition* is

$$\boxed{(AB)' \equiv B'A'}.$$

□ Note that it says that

$$(\overline{AB})^T \equiv \overline{B}^T \overline{A}^T$$

or equivalently, by transposition,

$$\overline{AB} \equiv \overline{A}\,\overline{B}. \qquad (*)$$

Let $C = \left[\gamma_{ji} \right] := AB$. Then

$$\gamma_{ji} = \sum_{k=1}^{p} \alpha_{jk} \beta_{ki} .$$

Now we have

$$\overline{\alpha + \beta} \equiv \overline{\alpha} + \overline{\beta} , \quad \overline{\alpha\beta} \equiv \overline{\alpha}\,\overline{\beta}$$

for $\alpha, \beta \in C$. And, more generally,

$$\overline{\alpha_1+\alpha_2+\ldots+\alpha_p} \equiv \overline{\alpha}_1+\overline{\alpha}_2+\ldots+\overline{\alpha}_p$$

for $\alpha_1, \alpha_2, \ldots, \alpha_p \in \mathbb{C}$. Thus

$$\overline{\gamma}_{ji} \equiv \overline{\sum_{k=1}^{p} \alpha_{jk}\beta_{ki}} \equiv \sum_{k=1}^{p} \overline{\alpha_{jk}\beta_{ki}}$$

$$\equiv \sum_{k=1}^{p} \overline{\alpha}_{jk}\overline{\beta}_{ki},$$

showing that $\overline{C} = \overline{AB} = \overline{A}\,\overline{B}$, i.e., showing that (*) is true. ■

Euclidean inner product and norm for $\mathbb{F}^n$ ($\mathbb{F} = \mathbb{R}$ *or* $\mathbb{C}$).

The *Euclidean inner product of* the vectors $x, y \in \mathbb{F}^n$ is

$$<y, x> = <y, x>_2 := y'x$$

$$= \underline{\qquad\qquad}\Big| =$$

$$= \begin{bmatrix} \eta_1' & \eta_2' & \ldots & \eta_n' \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix}$$

$$= \sum_{k=1}^{n} \eta_k' \xi_k = \sum_{k=1}^{n} \overline{\eta}_k \xi_k$$

$$= y' * x \quad \text{in matlab.}$$

Note that

$$x'x = \sum_{k=1}^{n} \overline{\xi}_k \xi_k = \sum_{k=1}^{n} |\xi_k|^2 \geq 0,$$

with equality if and only if $x = 0$.

The *Euclidean norm* of $x \in \mathbb{F}^n$ is

$$\|x\| = \|x\|_2 := \text{sqrt}(x'x)$$

$$= \left(\sum_{k=1}^{n} |\xi_k|^2\right)^{1/2}$$

$$= \text{norm}(x) \quad \text{in matlab.}$$

There are more general inner products and norms for F^n, but these Euclidean ones, $y'x$ and $\sqrt{x'x}$, are representative and good enough for our basic purposes. The Euclidean norm of x is sometimes also called the 2-norm of x or the (Euclidean) *length* of x. The latter should not be confused with *matlab's length* n of an n-vector x.

The primary use of vector norms is to determine if the elements of the vector are large or small. For instance if we solve $Ax = b$ by some algorithm then, even in "nice" cases, the numerical solution, call it fl(x), cannot be expected to agree with the true solution x due to rounding errors. The norm $\| fl(x) - x \|_2$ gives us a feel for how good, or how bad, fl(x) approximates x. If x itself is large, or small, then it is more appropriate to work with the relativized error norm $\frac{\| fl(x) - x \|_2}{\| x \|_2}$. Thus the use of norms in this way *condenses many errors*, i.e., the errors in *each element* of the computed x, into a *single number*. See, for example, the diary gfsf. The Euclidean norm $\| x \| = \| x \|_2$ (and all general norms) satisfy the *homogeneity property*

$$\boxed{\| \alpha x \| \equiv | \alpha | \, \| x \|}.$$

□ For the definition of $\| x \| = \| x \|_2$ and the reverse order rule for conjugate transposition give

$$\begin{aligned} \| \alpha x \|^2 &= (\alpha x)'(\alpha x) = x'\alpha' \alpha x \\ &= x'\bar{\alpha}\alpha x = x' |\alpha|^2 x \\ &= |\alpha|^2 x'x = |\alpha|^2 \, \| x \|^2 \end{aligned}$$

and the result follows by taking square roots. ■

In particular we have

$$\boxed{|\alpha \xi| \equiv |\alpha| |\xi|, \ \alpha, \xi \in C}.$$

A vector $u \in F^n$ is a *unit vector* (with respect to the Euclidean norm) if

$$\| u \|_2 = 1.$$

For instance the axis vectors e_i are very special unit vectors.

Any nonzero vector $x \in F^n$ can be *normalized* to become a unit vector: $u = \frac{x}{\| x \|_2}$.

□ For $\alpha = \frac{1}{\| x \|_2} > 0$ is a scalar and $\| u \|_2 = \alpha \, \| x \|_2 = 1$. ■

Two vectors, $x, y \in F^n$ are *orthogonal*, or *perpendicular* (with respect to the Euclidean inner product) if

$$y'x = 0.$$

Of course this is equivalent with $x'y = (y'x)' = 0$. If x and y are orthogonal we sometimes write $x \perp y$, or $y \perp x$.

Examples.

1. The axis vectors e_i and e_j are orthogonal if $i \neq j$.
2. The columns of the real rotation matrix

$$R_\theta = \begin{bmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{bmatrix}, \quad \gamma = \cos\theta, \quad \sigma = \sin\theta,$$

are orthogonal since

$$\begin{bmatrix} \gamma & \sigma \end{bmatrix} \begin{bmatrix} -\sigma \\ \gamma \end{bmatrix} = -\gamma\sigma + \sigma\gamma = 0.$$

These columns are also unit vectors since

$$\gamma^2 + \sigma^2 = 1.$$

3. The columns of the matrix

$$A_z = \begin{bmatrix} x & -y \\ y & x \end{bmatrix}$$

associated with the complex number

$$z = x + iy$$

are orthogonal:

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} -y \\ x \end{bmatrix} = -xy + yx = 0.$$

The common 2-norms of these columns is

$$\sqrt{x^2 + y^2} = |z|.$$

If $z \neq 0$ we can normalize these columns to get the *polar factorization* of A_z:

$$A_z = |z| R_\theta$$

with R_θ as in #2 with

$$\gamma = \frac{x}{|z|} = \cos\theta, \quad \sigma = \frac{y}{|z|} = \sin\theta.$$

We have

$$\begin{aligned}\arg z &:= \textit{argument} \text{ of } z\\ &:= \theta.\end{aligned}$$

The argument θ is determined only up to additive integer multiples of 2π. The *principal* value of θ should be taken to satisfy $-\pi < \theta \leq \pi$.

Now, for *two* nonzero complex numbers z_0 and z_1, we have

$$A_{z_0} A_{z_1} = |z_0||z_1| R_{\theta_0} R_{\theta_1}.$$

By problem 7 in the section on Complex Numbers, the left side is

$$A_{z_0 z_1} = |z_0 z_1| R_\theta, \quad \theta := \arg(z_0 z_1)\ .$$

But $|z_0 z_1| = |z_0||z_1| > 0$ (page 31) so we have

$$R_\theta = R_{\theta_0} R_{\theta_1}\ .$$

In fact these are just the *trigonometric addition formulas*

$$\begin{aligned}\cos(\theta_0 + \theta_1) &= \cos\theta_0 \cos\theta_1 - \sin\theta_0 \sin\theta_1\\ \sin(\theta_0 + \theta_1) &= \sin\theta_0 \cos\theta_1 + \cos\theta_0 \sin\theta_1\ .\end{aligned}$$

Thus $\theta = \theta_0 + \theta_1$, that is

$$\boxed{\arg(z_0 z_1) = \arg z_0 + \arg z_1}\,,$$

up to additive integer multiples of 2π. The polar factorization $A_z = |z| R_\theta$ is just a disguised form of the polar factorization

$$z = re^{i\theta}, \quad r = |z|, \quad \theta = \arg z,$$

of z itself. The matrix R_θ corresponds with the complex number

$$\boxed{e^{i\theta} := \operatorname{cis}\theta := \cos\theta + i\sin\theta}\,.$$

The trigonometric addition formulas simply state that

$$\boxed{e^{i(\theta_0+\theta_1)} = e^{i\theta_0}\, e^{i\theta_1}}\,.$$

This is the *law of exponents* for the *exponential function of an imaginary argument.*

In this example we used the matrix connection $z \leftrightarrow A_z$ to develop the polar factorization of a complex number and its properties. *More general* polar factorizations of $n \times n$ matrices *are important.* They are easy consequences of the "singular value decomposition."

4. The columns of the *quaternion*

$$Q = \begin{bmatrix} a & -\bar{b} \\ b & \bar{a} \end{bmatrix}, \quad a, b \in \mathbb{C}$$

are orthogonal:

$$\begin{bmatrix} -b & a \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = -ba + ab = 0.$$

They have common (squared) Euclidean norm:

$$\begin{bmatrix} \bar{a} & \bar{b} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \bar{a}a + \bar{b}b = |a|^2 + |b|^2,$$

$$\begin{bmatrix} -b & a \end{bmatrix} \begin{bmatrix} -\bar{b} \\ \bar{a} \end{bmatrix} = \bar{b}b + a\bar{a} = |a|^2 + |b|^2.$$

Problem 7.

a. Compute $y'x$ where

$$x := \begin{bmatrix} 1+i \\ -1 \\ 2-i \end{bmatrix}, \quad y := \begin{bmatrix} 3 \\ 1-i \\ 2 \end{bmatrix}.$$

Are x and y orthogonal?

b. Compute the 2-norms of x and y. Normalize x and y to get unit vectors u and v.

Problem 8.

a. Show that the columns of

$$H = \text{mxhadamard}(4)$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & -1 \end{bmatrix}$$

are orthogonal with each other and that their common 2-norms are 2. Hint: Compute $H'H$. Normalize the columns of H to obtain a matrix G with all columns unit vectors. Show that $G'G = I_4 = GG'$.

b. Same problem for

$$W = \text{mxidft}(4)$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

6. Frobenius and 2-norms of $A \in F^{n \times m}$.

We must also be able to determine if all elements of a matrix are "small" or if some are "large." This involves the notion of *matrix norms.*

The 2-*norm* of $A \in F^{n \times m}$ is

$$\| A \|_2 := \max_{x \neq 0} \frac{\| Ax \|_2}{\| x \|_2}$$

$$= \text{norm}\,(A) \quad \text{in matlab.}$$

This matrix norm is the most basic and elegant of matrix norms but it is not easy to compute. In fact

$$\| A \|_2 = \sigma_1(A)$$

is the largest "singular value" of A. More about "singular values" in advanced courses. They are *very important.* We merely give a geometric interpretation of $\| A \|_2$ here. By the homogeneity of the vector 2-norm we also have

$$\| A \|_2 := \max_{\| u \|_2 = 1} \| Au \|_2$$

Now the set

$$\left\{ u \in F^m : \ \| u \|_2 = 1 \right\}$$

is the *unit sphere* in F^m. It is transformed by A into the set

$$\left\{ Au : u \in F^m, \ \| u \|_2 = 1 \right\}$$

in $\mathbb{F}^n$. (This set is actually the boundary of an ellipsoid in $\mathbb{F}^n$ but that is not crucial here.) In any case, $\|A\|_2$ represents the "*maximum enlargement*" of the transformation of the unit sphere in $\mathbb{F}^m$ by A. For $m = n = 1$ we have $A = \alpha$ a scalar and it is clear that $\|A\|_2 = |\alpha|$. For general m and n computing $\|A\|_2$ *properly*, from a numerical point of view, involves finding the largest "eigenvalue" of the Jordan-Lanczos matrix

$$\mathcal{A} := \begin{bmatrix} 0 & A' \\ A & 0 \end{bmatrix}$$

associated with A.

The *Frobenius norm* of A, $\|A\|_F$, is easily computable and is just as powerful as $\|A\|_2$. We have

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix} = \begin{bmatrix} \alpha_{ji} \end{bmatrix}, \qquad i = 1, 2, \ldots, m, \quad j = 1, 2, \ldots, n,$$

and the associated "supervector"

$$a := \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \in \mathbb{F}^{mn}$$

$$= A(:) \quad \text{in matlab,}$$

obtained by stacking the columns of A. The *Frobenius norm* of A is simply

$$\begin{aligned} \|A\|_F &= \|a\|_2 \\ &= \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |\alpha_{ji}|^2 \right)^{1/2} \\ &= \text{norm}(A, \text{'fro'}) \\ &= \text{norm}(A(:)) \quad \text{in matlab} \\ &= \text{normf}(A) \quad \text{in Gragg's codes.} \end{aligned}$$

There are two other ways to write $\|A\|_F$.

The *trace* of a (square) matrix B is the sum of its diagonal elements. If $B \in \mathbb{F}^{n \times n}$ then

$$\text{trace } B := \sum_{k=1}^{n} \beta_{kk} .$$

Clearly,

$$\text{trace} : F^{n\times n} \to F$$

is a linear transformation:

$$\text{trace}\,(\alpha A + \beta B) \equiv \alpha \text{ trace } A + \beta \text{ trace } B.$$

Now we have

$$A'A = \begin{bmatrix} a_1' \\ a_2' \\ \vdots \\ a_m' \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix}$$

$$= \begin{bmatrix} a_j' a_i \end{bmatrix}, \qquad i = 1, 2, \ldots, m, j = 1, 2, \ldots, m\,.$$

Thus

$$\text{trace } A'A = \sum_{i=1}^{m} a_i' a_i = \sum_{i=1}^{m} \| a_i \|_2^2$$

is the sum of the squared 2-norms of the columns of A. But we have

$$\| a_i \|_2^2 = \sum_{j=1}^{n} |\alpha_{ji}|^2$$

so

$$\text{trace } A'A = \sum_{i=1}^{m} \sum_{j=1}^{n} |\alpha_{ji}|^2 = \| A \|_F^2\,.$$

It is also clear that $\| A \|_F = \| A' \|_F$ so we have

$$\| A \|_F = (\text{trace } A'A)^{1/2} = (\text{trace } AA')^{1/2}.$$

The latter equality can also be gotten by using

$$AA' = \sum_{i=1}^{m} a_i a_i'$$

and the linearity of trace.

It is known that

$$\| A \|_2 \le \| A \|_F \le \sqrt{\ell}\, \| A \|_2, \quad \ell := \min\{m, n\}\,.$$

Thus if (ℓ is fixed and) $\| A \|_2$ is "tiny" (or "huge") then so is $\| A \|_F$, and vice versa.

Problem 9.

Use $AA' = \sum_{1}^{m} a_i a_i'$ and the linearity of trace to show that trace $AA' = \| A \|_F^2$.

Problem 10.

For 100 random 50×50 matrices, A = rand (50) or A = random (50) (Gragg's code) compute the smallest and largest of the ratios

$$\frac{\| A \|_F}{\| A \|_2} = \frac{\text{normf(A)}}{\text{norm (A)}}$$

thus confirming the (unproved!) inequality at the bottom of page 37, with $\sqrt{\ell} = \sqrt{50} \doteq 7.07$.

Problem 11.

The 2-*condition number* of an $n \times n$ "nonsingular" matrix A is

$$\kappa_2(A) := \| A \|_2 \; \| A^{-1} \|_2$$

$$= \text{norm (A)} * \text{norm (inv (A))} = \text{cond (A)} \quad \text{in matlab.}$$

After we study "inverses" ($A^{-1} = \text{inv (A)}$) we'll show how $\kappa_2(A)$ governs the sensitivity of the solution x of $Ax = b$ to changes in the right side b. But for now we want to relate $\kappa_2(A)$ with the more readily computable "Frobenius condition number"

$$\kappa_F(A) := \| A \|_F \; \| A^{-1} \|_F$$

$$= \text{normf(A)} * \text{normf(inv (A))} \quad \text{with Gragg's normf.}$$

a. Use the (unproved!) inequality above to show that, for $n \times n$ matrices A,

$$1 \le \frac{\kappa_F(A)}{\kappa_2(A)} \le n \, .$$

b. For 100 random matrices, as in problem 10, compute the smallest and largest of the ratios $\frac{\kappa_F(A)}{\kappa_2(A)}$, and thus confirm experimentally the inequality you proved in a.

7. Solving (lower) triangular systems.

We work with the example

$$Lx = b : \begin{bmatrix} 1 & & & \\ 2 & 5 & & \\ 3 & 6 & 8 & \\ 4 & 7 & 9 & 10 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \xi_4 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ 5 \\ -4 \end{bmatrix}$$

We want to solve for

$$x = \begin{bmatrix} \xi_1 & \xi_2 & \xi_3 & \xi_4 \end{bmatrix}^T.$$

We first proceed directly.

The system of equations is

$$\begin{aligned} 1\xi_1 &= 1 \\ 2\xi_1 + 5\xi_2 &= -3 \\ 3\xi_1 + 6\xi_2 + 8\xi_3 &= 5 \\ 4\xi_1 + 7\xi_2 + 9\xi_3 + 10\xi_4 &= -4 \end{aligned}$$

We *f*(orward)-*solve* it.

$$\xi_1 = \frac{1}{1} = 1$$

$$\xi_2 = \frac{-3 - 2\xi_1}{5}$$

$$= \frac{-3-2}{5} = -1$$

$$\xi_3 = \frac{5 - 3\xi_1 - 6\xi_2}{8}$$

$$= \frac{5 - \begin{bmatrix} 3 & 6 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix}}{8}$$

$$= \frac{5 - \begin{bmatrix} 3 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix}}{8}$$

$$= \frac{5-(-3)}{8} = 1,$$

$$\xi_4 = \frac{-4 - 4\xi_1 - 7\xi_2 - 9\xi_3}{10}$$

$$= \frac{-4 - \begin{bmatrix} 4 & 7 & 9 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix}}{10}$$

$$= \frac{-4 - \begin{bmatrix} 4 & 7 & 9 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}}{10}$$

$$= \frac{-4-6}{10} = -1 \, .$$

This might be called the *scalar product form* of f-solving lower triangular systems. The solution is

$$x = \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix}^T$$

as (was prearranged and) is easily checked by matrix-vector multiplication:

$$Lx = \begin{bmatrix} 1 & & & \\ 2 & 5 & & \\ 3 & 6 & 8 & \\ 4 & 7 & 9 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} 1 + \begin{bmatrix} 0 \\ 5 \\ 6 \\ 7 \end{bmatrix} (-1) + \begin{bmatrix} 0 \\ 0 \\ 8 \\ 9 \end{bmatrix} 1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 10 \end{bmatrix} (-1)$$

$$= \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 0 \\ -5 \\ -6 \\ -7 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 8 \\ 9 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ -10 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ -3 \\ -3 \\ -3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 8 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ 5 \\ -4 \end{bmatrix} = b$$

(We did some "distributed computing" here!)

Using matlab-like notation, in general, at the kth step we must solve

$$L(k,\ 1:k-1)x(1:k-1)+L(k,\ k)x(k)=b(k)$$

for x(k):

$$x(k)=\frac{b(k)-L(k,\ 1:k-1)*x(1:k-1)}{L(k,\ k)}.$$

Note that $x(1:k-1)$ *must be a column*. Because of this, and to make the for loop go faster we initially *reserve* a column for x. Also, this command will not work for $k=1$ since $L(1,\ 1:0)$ and $x(1:0)$ are *empty* ([ ] in matlab). We handle the case $k=1$ separately. Since the "vector" $i:=1:k-1$ of indices occurs more than once we give it a name.

```
function x = gfsfsp(L, x)
n = order (L);        x = zeros(n, 1);
x(1) = b(1)/L(1, 1);
for k = 2 : n
   i = 1 : k − 1;
   x(k) = (b(k) − L(k, i) * x(i))/L(k, k);
end
```

Order is a "Gragg-code" which gives the *order* of a *square* matrix or an error message if the matrix is not square. If $n<2$ the for loop is not executed.

We wish to write some *column oriented* codes for f-solving. Transliterations of such codes into a "production" language like fortran will be more efficient for large matrices because such matrices are stored by columns. Also this approach is more like the techniques we use for *factoring* $A=LU$ = . Our system is

$$Lx=b:\quad \left[\begin{array}{c|ccc} 1 & & & \\ \hline 2 & 5 & & \\ 3 & 6 & 8 & \\ 4 & 7 & 9 & 10 \end{array}\right]\left[\begin{array}{c} \xi_1 \\ \hline \xi_2 \\ \xi_3 \\ \xi_4 \end{array}\right]=\left[\begin{array}{c} 1 \\ \hline -3 \\ 5 \\ -4 \end{array}\right].$$

If $n=1$ we have $\xi_1=\frac{\beta_1}{\lambda_{11}}$, and we are done.

In general we *partition* a lower triangular system as

$$\begin{bmatrix} \lambda & \\ \ell & M \end{bmatrix} \begin{bmatrix} \xi \\ y \end{bmatrix} = \begin{bmatrix} \beta \\ c \end{bmatrix} \qquad (*)$$

say. L *must* have all its diagonal elements nonzero. If $n = 1$ the matrices ℓ, M, y and c are all empty. The system $(*)$ is equivalent with

$$\lambda\xi = \beta, \quad \ell\xi + My = c.$$

Thus we compute

$$\xi = \frac{\beta}{\lambda},$$

and then obtain the *reduced system*

$$My = c - \ell\xi =: d$$

which we must ultimately solve for y. We then *repeat the reduction* until the reduced system becomes one equation in one unknown. Thus it is like $(*)$ but with ℓ, M, y and c empty.

For our example, with the indicated partitioning, the first equation is

$$1\xi = 1 \quad \text{so} \quad \xi_1 = \xi = \frac{1}{1} = 1$$

The right side of the reduced system is

$$d = c - \ell\xi = \begin{bmatrix} -3 \\ 5 \\ -4 \end{bmatrix} - \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} 1 = \begin{bmatrix} -5 \\ 2 \\ -8 \end{bmatrix}.$$

The reduced system is

$$\left[\begin{array}{c|cc} 5 & & \\ \hline 6 & 8 & \\ 7 & 9 & 10 \end{array}\right] \left[\begin{array}{c} \xi_2 \\ \hline \xi_3 \\ \xi_4 \end{array}\right] = \left[\begin{array}{c} -5 \\ \hline 2 \\ -8 \end{array}\right]$$

The first equation is

$$5\xi = -5 \quad \text{so} \quad \xi_2 = \xi = \frac{-5}{5} = -1.$$

The right side of the next reduced system is

$$d = c - \ell\xi = \begin{bmatrix} 2 \\ -8 \end{bmatrix} - \begin{bmatrix} 6 \\ 7 \end{bmatrix}(-1) = \begin{bmatrix} 8 \\ -1 \end{bmatrix}.$$

The whole reduced system is

$$\left[\begin{array}{c|c} 8 & \\ \hline 9 & 10 \end{array}\right] \left[\begin{array}{c} \xi_3 \\ \hline \xi_4 \end{array}\right] = \left[\begin{array}{c} 8 \\ \hline -1 \end{array}\right].$$

The first equation is

$$8\xi = 8 \qquad \text{so} \qquad \xi_3 = \xi = \frac{8}{8} = 1\,.$$

The right side of the final reduced system is

$$d = c - \ell\xi = -1 - 9 \cdot 1 = -10.$$

The final reduced system is

$$10\xi_4 = -10 \qquad \text{so} \qquad \xi_4 = -1.$$

Of course the solution is the same as before. In fact, if the scalar products L(k, 1 : k − 1) ∗ x(1 : k − 1) are computed "from left to right" the computations are exactly the same as before. That is they even have the same rounding errors!

This can be turned into matlab code by using the "template"

$$\begin{array}{cc} & \begin{array}{cc} k & \end{array} \\ k & \begin{bmatrix} \lambda & \\ \ell & M \end{bmatrix} \end{array} \begin{bmatrix} \xi \\ y \end{bmatrix} = \begin{bmatrix} \beta \\ c \end{bmatrix}.$$

Here

$$\lambda = L(k, k), \qquad \xi = x(k),$$
$$\ell = L(j, k) \qquad M = L(j, j)\,,$$

with

$$j = k + 1 : n\,.$$

The right side, which is continually changing, is stored in b(k : n) so

$$\beta = b(k), \quad c = b(j)\,.$$

Thus

$$x(k) = \frac{b(k)}{L(k, k)}$$

and the statement $d = c - \ell\xi$ becomes the *replacement*

$$b(j) = b(j) - L(j, k) * x(k) .$$

Our "zeroth" version of the code gfsf is thus

```
function x = gfsf0(L, b)
n = order (L); x = zeros (n, 1);
for k = 1 : n-1
    x(k) = b(k)/L(k, k);
    j = k+1 : n;
    b(j) = b(j) - L(j, k) * x(k);
end
x(n) = b(n)/L(n, n);
```

Note that the empty matrices occur at the last step here. Unfortunately matlab does not overlook statements like [ ] = [ ] which would occur at the last statement of the for loop for k = n. It's good practice to reference subscripted variables as little as possible. This probably doesn't matter so much with matlab but it is good style. Also, we can initialize x as b and work only with the single vector x. In fortran x and b would be made "equivalent" so only one vector would be stored.

Our final version of the (nonrecursive) code is

```
function x = gfsf(L, b)
n = order (L);      x = b;
for k = 1 : n-1
   t = x(k)/L(k, k);   x(k) = t;
   j = k+1 : n;   x(j) = x(j) - L(j, k) * t;
end
x(n) = x(n)/L(n, n);
```

This is the way that good code is written, in stages. Here we used *only* two stages! We much prefer this code over gfsfsp. We should have used the replacement x = b in it too!

Codes using *recursive calls* are even easier. (They are for "lazy people"!) We build our code gfsfr. Recall that, given the system

$$\begin{array}{ccc} & 1 & \\ 1 & \left[\begin{array}{cc}\lambda & \\ \ell & M\end{array}\right] & \left[\begin{array}{c}\xi \\ y\end{array}\right] = \left[\begin{array}{c}\beta \\ c\end{array}\right],\end{array}$$

we compute

$$\xi = \frac{\beta}{\lambda}$$

and then solve the *reduced system*

$$My = c - \ell\xi$$

for y. But we can replace M by L (i.e., we can call it L) and b by $c - \ell\xi$. Then we have $y = \text{gfsfr}(L, b)$. Thus gfsfr calls itself recursively and the order is automatically, recursively, reduced to $n = 1$. This is the only case we have to compute, apart from "downdating" L and b. If we initially put $x = b(1)/L(1, 1)$ then we are done if $n = 1$. Otherwise, we can avoid introducing y by making the replacement $x = [x ; \text{gfsfr}(L, L)]$, after "downdating" L and b.

```
function x = gfsfr(L, b)
n = order (L);        x = b(1)/L(1, 1);
if n > 1
   j = 2 : n;         b = b(j) - L(j,1) * x;
   L = L(j, j);       x = [x; gfsfr(L, b)];
end
```

This code delivers the *same computational results* as gfsf. It is essentially the same code as gfsf. The compiler does the work of looping. But it is about *three times slower* than gfsf! Considering the "strassen experience" that's not too bad!

Problem 12. Write codes analogous with gfsf and gfsfr (call them gfsb and gfsbr) for *backsolving* upper triangular systems $Ux = b$.

Appendix. Some simple codes which use recursive calls.

function f = factorial (n)

f is n FACTORIAL. That is f is the product of the integers 1, 2, 3, ..., n. If n = 0 this is the empty product 1. The function factorial is probably the simplest instance of a program which uses recursive calls. It assumes that n is a nonnegative integer.

Copyright (c) 20 July 1991 by Bill Gragg. All rights reserved.

```
factorial calls factorial.

begin factorial
    if n <= 0
       f = 1;
    else
       f = n*factorial (n-1);
    end
end factorial
```

```
function H = mxhadamard (n)
H is a HADAMARD MATRIX of order n.  n must be power of two.
Copyright (c) 6 January 1991 by Bill Gragg.  All rights reserved.
mxhadamard calls mxhadamard.

begin mxhadamard
    if n < 2
       H = 1;
    else
       H = mxhadamard (n/2);      H = [H  H;  H  -H];
    end
end mxhadamard
```

Computing $\|A\|_2$, for $A \in \mathbb{R}^{2\times 2}$.

It's not easy to compute $\|A\|_2$, even for real 2x2s. Matlab's norm(A) does it for you, even for $A \in \mathbb{C}^{n\times m}$. More generally, svd(A) computes the singular values of A. For $A \in \mathbb{R}^{2\times 2}$ these are

$$\sigma_1(A) := \max_{\|x\|_2=1} \|Ax\|_2 =: \|A\|_2,$$

$$\sigma_2(A) := \min_{\|x\|_2=1} \|Ax\|_2 .$$

The condition number of A is

$$\operatorname{cond} A = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$$

$$= \frac{\sigma_1(A)}{\sigma_2(A)}, \quad A \in \mathbb{R}^{2\times 2}.$$

We show how to compute these numbers, and give an example, for $A \in \mathbb{R}^{2\times 2}$. These concepts are extremely important and are covered in more detail in

Here we go. We can work with the square:

$$\|Ax\|_2^2 = (Ax)^T Ax = x^T A^T A x = x^T B x$$

with

$$B := A^T A = \begin{bmatrix} \alpha_{11} & \alpha_{21} \\ \alpha_{12} & \alpha_{22} \end{bmatrix} \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix}$$

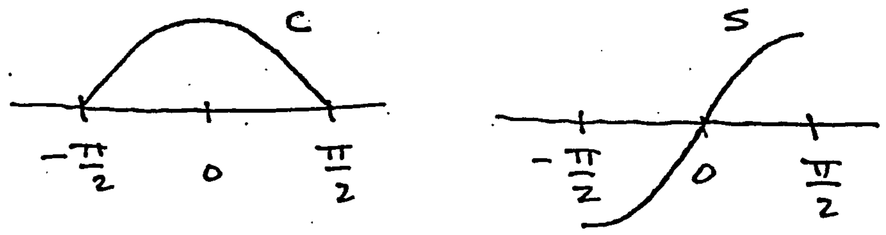
$$= \begin{bmatrix} \alpha_{11}^2 + \alpha_{21}^2 & \alpha_{11}\alpha_{12} + \alpha_{21}\alpha_{22} \\ \alpha_{11}\alpha_{12} + \alpha_{21}\alpha_{22} & \alpha_{12}^2 + \alpha_{22}^2 \end{bmatrix}$$

$$= \begin{bmatrix} \beta_{11} & \beta_{21} \\ \beta_{21} & \beta_{22} \end{bmatrix} \quad \text{(symmetric!)}.$$

Want the max and min of $x^T B x$ over x with $\|x\|_2^2 = x^T x = 1$. Let

$$x = \begin{bmatrix} c \\ s \end{bmatrix}, \quad c = \cos\theta, \; s = \sin\theta.$$

We can insist that $-\frac{\pi}{2} < \theta \le \frac{\pi}{2}$ (why?). Then $c \geq 0$:



Always $c^2 + s^2 \equiv 1$. Now

$$x^T B x = [c \ s] \begin{bmatrix} \beta_{11} c + \beta_{21} s \\ \beta_{21} c + \beta_{22} s. \end{bmatrix}$$

$$= c(\beta_{11} c + \beta_{21} s) + s(\beta_{21} c + \beta_{22} s)$$

$$= \beta_{11} c^2 + 2\beta_{21} cs + \beta_{22} s^2,$$

a <u>function of</u> θ. By Calculus we need

$$\frac{d}{d\theta} x^T B x = -2\beta_{11} cs + 2\beta_{22} cs + \\ + 2\beta_{21}(c^2 - s^2)$$

$$= 2\left[-(\beta_{11} - \beta_{22}) cs + \beta_{21}(c^2 - s^2)\right] = 0.$$

1. If $\beta_{21} = 0$ but $\beta_{11} \neq \beta_{22}$ we must have $cs = 0$, that is $\theta = 0$ or $\theta = \frac{\pi}{2}$. This gives the two optimal vectors

$$x_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

with associated values

$$\lambda_1 := x_1^T B x_1 = \beta_{11} = \alpha_{11}^2 + \alpha_{21}^2,$$

$$\lambda_2 := x_2^T B x_2 = \beta_{22} = \alpha_{12}^2 + \alpha_{22}^2.$$

This also works when $\beta_{11} = \beta_{22}$.

2. Now suppose that $\beta_{21} \neq 0$.

have

$$\frac{c^2 - s^2}{cs} = \boxed{\frac{\beta_{11} - \beta_{22}}{\beta_{21}} =: 2\beta}. \qquad (1)$$

Let

$$t := \tan\theta = \frac{s}{c}.$$

Then this is

$$\frac{1}{t} - t = 2\beta,$$

or

$$t^2 + 2\beta t - 1 = 0,$$

a quadratic equation in t with the <u>two</u> solutions

$$t_1 = \boxed{t := -\beta + \sqrt{1+\beta^2} > 0},$$
$$t_2 = -\frac{1}{t} = -\left(\beta + \sqrt{1+\beta^2}\right) < 0.$$

With

$$t = \frac{s}{c}, \quad c > 0, \quad s > 0,$$

we have

$$1 = c^2 + s^2 = c^2(1+t^2)$$

so

$$\boxed{c = \frac{1}{\sqrt{1+t^2}}, \quad s = \frac{t}{\sqrt{1+t^2}}}. \qquad (2)$$

Now if

$$t_2 = \frac{s_2}{c_2}, \quad c_2 > 0,$$

then from

$$t_2 = -\frac{c}{s}$$

we have that

$$c_2 = s, \quad s_2 = -c.$$

Thus

$$t_1 \leftrightarrow x_1 = \begin{bmatrix} c \\ s \end{bmatrix}, \quad t_2 \leftrightarrow x_2 = \begin{bmatrix} s \\ -c \end{bmatrix}.$$

The optimal vectors x_1 and x_2 are <u>orthogonal</u>!

The optimal <u>values</u> of $x^T B x$ are

$$\lambda_1 = x_1^T B x_1 = \beta_{11} c^2 + 2\beta_{21} cs + \beta_{22} s^2,$$
$$\lambda_2 = x_2^T B x_2 = \beta_{11} s^2 - 2\beta_{21} cs + \beta_{22} c^2,$$

and since $c^2 + s^2 = 1$ then

$$\lambda_1 + \lambda_2 = \beta_{11} + \beta_{22}. \qquad (3)$$

If one of λ_1 or λ_2 is known the other can be computed using this formula.

Since $s^2 = 1 - c^2$ then

$$\lambda_1 = \beta_{22} + 2\beta_{21} cs + (\beta_{11} - \beta_{22}) c^2 .$$

By (2) and (1),

$$\lambda_1 = \beta_{22} + 2\beta_{21} \frac{t+\beta}{1+t^2} .$$

But

$$2 \frac{t+\beta}{1+t^2} = \frac{1}{t} ,$$

for this is the same as

$$2t(t+\beta) = 1 + t^2 ,$$

that is

$$t^2 + 2\beta t = 1 .$$

Thus

$$\boxed{\lambda_1 = \beta_{22} + \frac{\beta_{21}}{t}}$$

and by (3),

$$\boxed{\lambda_2 = \beta_{11} - \frac{\beta_{21}}{t}} .$$

Finally, the optimal values of $\|Ax\|_2 = \sqrt{x^T B x}$ are

$$\boxed{\begin{aligned} \sigma_1 &= \max\{\sqrt{\lambda_1}, \sqrt{\lambda_2}\} , \\ \sigma_2 &= \min\{\sqrt{\lambda_1}, \sqrt{\lambda_2}\} , \end{aligned}}$$

and

$$\boxed{\operatorname{cond} A = \frac{\sigma_1}{\sigma_2}}$$

Example. $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$.

$$B = A^T A = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix},$$

$$\beta = \frac{\beta_{11} - \beta_{22}}{2\beta_{21}} = \frac{1-5}{4} = -1,$$

$$t = -\beta + \sqrt{1+\beta^2} = 1 + \sqrt{2},$$

$$\lambda_1 = \beta_{22} + \frac{\beta_{21}}{t}$$

$$= 5 + \frac{2}{\sqrt{2}+1}\,\frac{\sqrt{2}-1}{\sqrt{2}-1}$$

$$= 5 + 2\,\frac{\sqrt{2}-1}{2-1} = 3 + 2\sqrt{2}$$

$$\doteq 4.4142$$

$$\lambda_2 = 6 - \lambda_1 = 3 - 2\sqrt{2}$$

$$\doteq 0.1716$$

$$\sigma_1 = \sqrt{\lambda_1} \doteq 2.4142$$

$$\sigma_2 = \sqrt{\lambda_2} \doteq 0.4142$$

$$\operatorname{cond} A = \frac{\sigma_1}{\sigma_2} \doteq 5.8284.$$

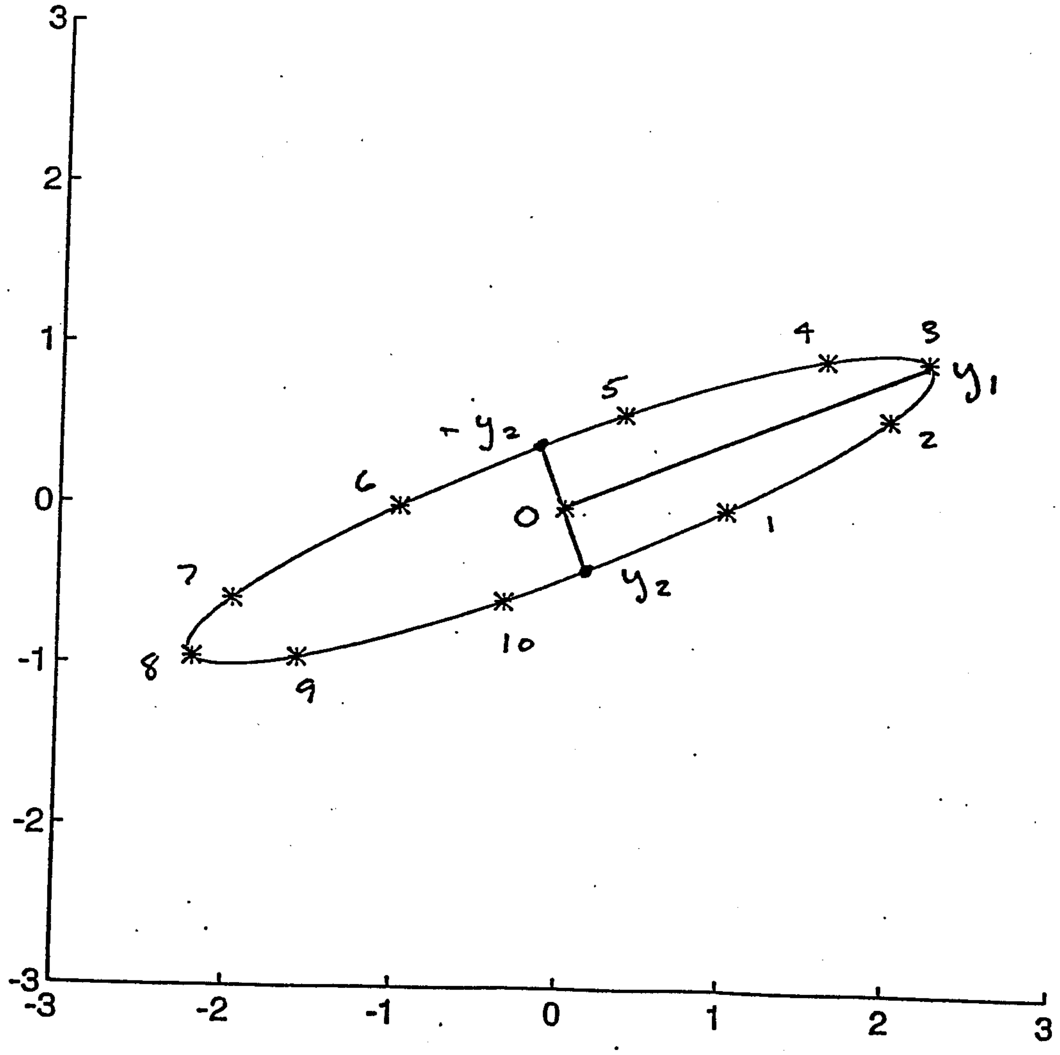
We also found, with a little (more) help from matlab, the optimal vectors . . .

$$x_1 = \begin{bmatrix} c \\ s \end{bmatrix} \doteq \begin{bmatrix} 0.3827 \\ 0.9239 \end{bmatrix},$$

$$x_2 = \begin{bmatrix} s \\ -c \end{bmatrix} \doteq \begin{bmatrix} 0.9239 \\ -0.3827 \end{bmatrix},$$

$$y_1 = Ax_1 \doteq \begin{bmatrix} 2.2304 \\ 0.9239 \end{bmatrix}, \quad \|y_1\|_2 = \sigma_1,$$

$$y_2 = Ax_2 \doteq \begin{bmatrix} 0.1585 \\ -0.3827 \end{bmatrix}, \quad \|y_2\|_2 = \sigma_2 .$$



3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3
0
1
2
3
4
5
6
7
8
9
10
y_1
y_2
$-y_2$